

Integrated Development Environment for Procedural Modeling

Master's Thesis

Institute of Computer Graphics and Knowledge Visualization
Graz University of Technology

Christoph Schinko
schinko@sbox.tugraz.at

Assessor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolf-Dietrich Fellner
Supervisor: Dipl.-Math. Torsten Ullrich

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am

.....

(Unterschrift)

Zusammenfassung

In dieser Arbeit wird eine Entwicklungsumgebung für prozedurales Modellieren sowie eine neue Programmiersprache mit dem Namen *Euclides* präsentiert. Die Syntax sowie die Sprachkonstrukte sind von der Programmiersprache *Pascal* abgeleitet. *Euclides* führt neue Konzepte wie Vektoren und Matrizen ein, die sehr oft in der Computergrafik Verwendung finden. Fehlerträchtige Techniken wie Pointer werden vermieden. Das führt zu einer leichtgewichtigen Programmiersprache, die man für prozedurales Modellieren verwenden kann. Die Hauptvorteile dieser Sprache sind ihre einsteigerfreundliche Syntax sowie die Unabhängigkeit der Engine was die Erzeugung von Output betrifft. So kann eine Vielzahl verschiedener Outputrepräsentationen implementiert werden.

Diese Arbeit gliedert sich in einen Teil der die Programmiersprache selbst beschreibt, sowie die implementierten Outputrepräsentationen. Zu Beginn wird eine Einleitung in die Sprachkonzepte gegeben, gefolgt von einer Beschreibung der syntaktischen und semantischen Validierung des Quelltextes. Das Ergebnis dieses Vorgangs ist eine Zwischenrepräsentation die als Basis für die Erzeugung von Output dient. Die Entwicklungsumgebung unterstützt eine Reihe von Outputrepräsentationen.

Eine erste Outputrepräsentation stellt generierter Text mit Verlinkungen dar. Er besteht aus *Euclides* Quelltext auf dem man eine Quelltextanalyse durchführen kann. Metainformationen sind im Quelltext eingebettet um eine erweiterte Ansicht auf den Quelltext zu ermöglichen. Verweise von Variablen und Konstanten im Quelltext zu einem Übersichtsbereich, der wiederum aus einer Liste aller im Quelltext vorkommenden Variablen und Konstanten besteht, steigern die Lesbarkeit.

Generierter *Euclides* Quelltext dient als zweite Outputrepräsentation. Er wird für die Umgestaltung des Quelltextes verwendet und unterstützt zum Beispiel das Umbenennen von Variablen sowie das Rücksetzen von Werten von Konstanten.

Eine dritte Outputrepräsentation wird durch die Generierung von Java Quelltext erreicht. Sie bietet Interpretation und Inspektion des Quelltextes durch die Verwendung einer Laufzeitumgebung die als Basis für den generierten Quelltext dient. Die Entwicklungsumgebung ist vorbereitet um zusätzliche Outputrepräsentationen sowie Zielsprachen zu unterstützen. Die Generative Modeling Language (GML) kann als eine weitere Zielsprache in Betracht gezogen werden. Das würde es dem System ermöglichen 3D Objekte darzustellen und zugleich als zusätzliches Interface für die GML Quelltext Generierung dienen.

Abstract

This master thesis presents an integrated development environment (IDE) for procedural modeling including a new programming language called *Euclides*. Its syntax and language constructs are derived from the programming language *Pascal*. *Euclides* introduces new concepts like vector and matrix data types needed very often in computer graphics while omitting error-prone techniques like pointers. This leads to a lightweight language to be used for procedural modeling. The main advantages of the language are its beginner friendly syntax and the independence of the engine concerning output generation. A variety of different output representations can be implemented.

The thesis is divided in a description of the programming language and the implemented output representations. Initially, an introduction to the language concepts is given, followed by a description of syntactical and semantical validation of created source code. The result of these processes is an intermediate representation which serves as a basis for output generation. The IDE supports a number of output representations.

A first output representation is a generated text incorporating links. It features *Euclides* source code to perform a source code analysis. Meta information is embedded in the source code in an attempt to obtain an extended view of the code. Links from variables and constants in the source code to an overview section consisting of a collection of all variables and constants used promote readability.

Generated *Euclides* code itself serves as a second output representation. It is used for the task of source code refactoring and supports for example renaming of variables and resetting values of constants.

A third output representation is acquired through the generation of Java code. It offers source code interpretation and inspection through a runtime environment which serves as a basis for the generated code. The IDE is prepared to support additional output representations and target programming languages. The generative modeling language (GML) can be thought of as another target language. This would enable the framework to output 3D objects and on the other hand would serve as an additional interface for generating GML code.

Acknowledgements

At first I would like to thank Torsten Ullrich for his continuous support and guidance. I would also like to thank Prof. Fellner for giving me the opportunity to write this thesis.

Furthermore I would like to thank the people at the Institute of Computer Graphics and Knowledge Visualization for a friendly working environment and many interesting discussions. Last but not least my gratitude goes to my family, especially my parents Jörg and Ilse Schinko, for their continuous support.

Contents

1	Introduction	1
2	Related Work	5
3	Designing a programming language	9
3.1	Euclides	13
3.2	Parsing Euclides	19
3.3	Validating syntax and semantics	23
3.4	Transpiling Euclides	31
4	Views	37
4.1	HTML Transpiler	37
4.2	Java Transpiler	38
5	Future Work	49
5.1	GML integration	49
5.2	Maya integration	49
5.3	New language constructs	50
6	Addendum	51
	List of Figures	67

Chapter 1

Introduction

This work describes an integrated development environment (IDE) for procedural modeling. The idea behind procedural modeling is to create a generalized representation of a model. Not the model itself, but the model class is described. For example not a specific tree is generated during procedural modeling, but a pattern of a tree. This is done by creating a set of rules used as a representation instead of directly working with geometric primitives on a rather low level of abstraction. Usually this set of rules can be parameterized to allow variability. This leads to a main advantage of procedural modeling. The ability to create models that are too complex for a person to build using *classical* modeling approaches like polygonal modeling. For example one can think of a forest consisting of a large number of different trees. Using such techniques, a single tree pattern in combination with different sets of parameters can be applied to model the entire forest. As a side effect the scene can be stored efficiently because only one rule is used for tree generation. There is no need to store large amounts of data in the form of polygonal meshes because they are generated out of rule sets. However, this is not generally applicable since it requires a certain amount of redundancy to be found in the nature of the object.

A possibility to describe a model procedurally is to use shape grammars. They consist of rules that define shapes and usually a generation engine that selects and processes rules. One can compare the learning curve to use shape grammars with that of other scripting languages.

Another approach is to use specialized programming languages to obtain a procedural description of a model. Havemann proposes a simple *stack based* programming language called Generative Modeling Language (GML). It incorporates principles of Postscript to form a language to describe 3D objects. This means that the GML employs a postfix notation and therefore requires a certain training period to understand the language constructs. However, results can be obtained without much effort as shown in Figure 1.1 and Listing 1.1.

```

1 /pillars { 1 3 15 {
2   offset 0 vector3 (0,0,1) 0.5 16 circle
3 /steinwand setcurrentmaterial
4 5 poly2doubleface
5 (0,4,1) extrude
6 } for } def
7
8 /offset 0 def
9 pillars
10 /offset 5 def
11 pillars
12
13 % roof
14 (7,2.5,4) (6.5,3,0) 2 quad
15 5 poly2doubleface
16 (0,1,1) extrude
17 [(0.5,-0.5,5) (0.5,5.5,5) (0.5,2.5,7)] 5 poly2doubleface
18 (0,13,0) extrude

```

Listing 1.1: This listing represents a simple GML example program generating the temple shown in Figure 1.1. One can see that only a few lines of code are needed to obtain useful output.

In contrast to *classical* modeling approaches several differences can be found:

- Procedural modeling is accompanied by a paradigm shift from objects to operations. Consequently the process of shape design becomes rule design.
- Values are replaced by parameters, which leads to a separation from data and operations.
- Complex objects can be created using special libraries that require only a few input parameters.

What all these concepts have in common is that compared to *classical* modeling approaches using sophisticated tools it is rather difficult to learn a special programming language or apply shape grammars. This work aims for beginners to be able to quickly produce reasonable results without a long training period.

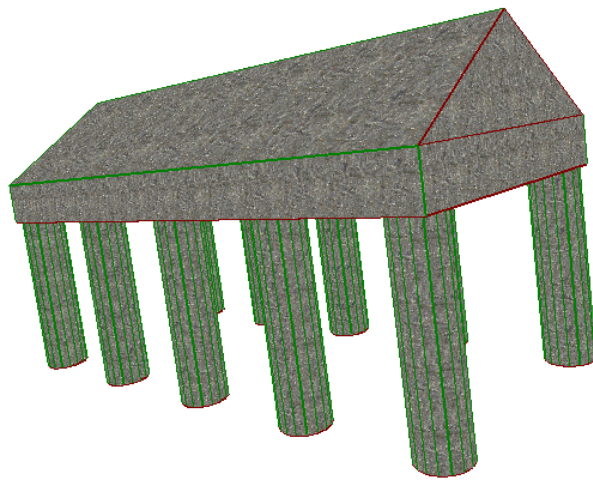


Figure 1.1: This figure illustrates the output of the GML example program shown in Listing 1.1. In order to identify the structures used to create the temple, it is printed with the control mesh.

Chapter 2

Related Work

Procedural techniques are a major topic in computer graphics nowadays. Describing a model with a set of rules and parameters is a very efficient way of storing it. The idea to trade processing effort for data size came up with the availability of modern hardware capable of profiting from that trade. Nevertheless, a lot of work has been done on this topic in the past as well.

Paoluzzi describes the use of a functional language in the context of geometric design programming [13]. The idea is to associate geometric shapes to generating functions and to pass geometric expressions as function parameters. This allows generation of methods describing geometric shape, as well as utilizing such methods for the purpose of modeling specific geometry. The generated objects are always consistent in geometry because the validity is guaranteed at a syntactical level.

Nevertheless, in modern CAD software products geometric validity is a subject when using parametric tools. For a given parametric model certain combinations of parameter values may not result in valid shapes. Hoffmann and Kim propose an algorithm [7] that computes valid parameter ranges for geometric elements in a plane, given a set of constraints.

In today's systems used for procedural modeling, grammars are often used as a set of rules to achieve a description. Early systems based on grammars were Lindenmayer systems [8], or L-systems for short. They were successfully applied to the process of modeling plants. Given a set of string rewriting rules, complex strings are created by applying these rules to simpler strings. This means that starting with an initial string, a predefined set of rules is applied to the string forming a new, possibly larger string. The L-systems approach 0 a parallel application of string rewriting rules in order to reflect their biological motivation. In order to use L-systems to model geometry an interpretation of the generated strings is necessary. Early results [4] used L-systems to determine branching of modeled plants. The modeling power of these early geometric interpretations of L-systems was limited to creating fractals and plant-like branching structures. This

lead to the introduction of parametric L-systems [17]. The idea is to associate numerical parameters with L-system symbols to address continuous phenomena which were not covered satisfactorily by L-systems alone. Later on, L-systems and shape grammars were successfully used in procedural modeling of cities [14]. Parish and Müller presented a system that, given a number of image maps as input, generates a street map including geometry for buildings. For that purpose L-systems have been extended to allow the definition of global objectives as well as local constraints. However, the use of procedurally generated textures to represent facades of buildings limits the level of detail in the results. In later work, Müller describes a system [12] to create detailed facades based on the split grammar called CGA shape.

Finkenzeller presented another approach for detailed building facades [3] based on a hierarchical description of an entire building. The user provides a coarse outline as well as a basic style of the building including distinguished parts and the system generates a graph representing the building. In the next step, the system traverses the graph and generates geometry for every element of the graph. This results in a detailed building facade with the limitation that it can handle common building structures only. Organic structures, inclined walls and details on the roofs are yet to be implemented.

Another modeling approach presented by Lipp et al. [9] following the notation of Müller [11] deals with the aspects of more direct local control of the underlying grammar by introducing visual editing. The idea is to allow modification of elements selected directly in a 3d-view, rather than editing rules in a text based environment. Therefore principles of semantic and geometric selection are combined as well as functionality to store local changes persistently over global modifications.

Lintermann et al. proposed a modeling method as well as a graphical user interface for the creation of natural branching structures [2]. A structure tree represents the modeling process and can be altered using specialized components describing geometry as well as structure. Another type of components can be used for defining global and partial constraints. Components are described procedurally using creation rules which include recursion. The generation of geometric data according to the structure tree is done via a tree traversal where the components generate their geometrical output.

The procedural modeling approach [5] proposed by Ganster et al. describes an integrated framework based on a visual language. The infix notation of the language requires the use of variables which are stored on a heap. A graph structure allows variable assignments to be performed by special nodes. Directed edges between nodes only define the order of execution, in contrast to a visual data flow pipeline where data is transported between the different stages. The framework allows fast creation of complex scenes with the limitation that geometry has to be modeled on a rather low level using polygon lists.

Havemann proposes a stack based language for creating polygonal meshes called GML. The postfix notation of the language is very similar to that of Adobe's Postscript. It allows the creation of high-level shape operators from low-level shape operators. The GML serves as a platform for a number of applications because it is extensible and comes with an integrated visualization engine. Havemann et al. presented a system for generative parametric design of Gothic window tracery [6]. Its complex geometric shape which consists of only a few basic geometric patterns is a property of Gothic architecture. The procedural approach relies on the combination of elementary operations and constructions to obtain an efficient parametric representation.

Another system presented by Mendez et al. combines semantic scene-graph markups with generative modeling in the context of generating semantic three dimensional models of underground infrastructure [10]. The idea is to connect a geospatial database and a rendering engine in order to create an interactive application. The GML is used for on-the-fly generation of procedural models in combination with a conventional scene graph with semantic markup. An augmented reality view of underground infrastructure like water or gas distribution systems serves as a demo application.

Day et al. presented a system combining polygonal and subdivision surface approaches in the context of modeling urban environments [1]. A modeler based on shells representing basic building units as well as a multi resolution surface modeling approach incorporating progressive meshes and combined boundary representations (B-reps) are discussed. Progressive meshes are a level of detail technique starting with a coarse description of the mesh while progressively refining it. A B-rep is a representation of a shape using its boundaries usually used in solid modeling. Ideas to combine both approaches consist of combining parts of the shell modeler with generative methods to increase efficiency.

Chapter 3

Designing a programming language

The motivation behind this work is to create a beginner friendly programming language for procedural modeling. In particular the requirements for such a language can be summarized as follows:

- The specification of the language regarding the syntax should be beginner friendly. A user with little or no experience in programming should be able to learn the language in a short time.
- Specific details like easy to handle vector and matrix data types needed for procedural modeling should be included in the specification. Techniques of a language that are error-prone, like pointers, can be left out. As Niklaus Wirth stated: “The most important decision in language design concerns what is to be left out.”

During an examination of current programming languages it became clear that none of them could comply with the requirements in a satisfactory way. Because of the fact that designing and implementing a new programming language would go beyond the scope of this work, the idea to design a syntax and transpile it into an already existing programming language was born. For that purpose it is reasonable to take an existing syntax as a reference. Each programming language is tailored for a specific purpose which leads to variations in syntax. Although, one can constrain the amount of programming languages by applying the previously stated requirements there is still a lot of choice left. The programming language Pascal represents the most important property mentioned in the requirements. It was designed to teach students structured, procedural programming and therefore has a rather intuitive syntax. Taking an already existing scripting engine as a starting point would lack the benefit of being able to independently choose the output representation. Consequently a good starting point for a language is a variation of the Pascal syntax.

In order to define the syntax of a programming language a grammatical description is needed. This description itself is written using ANOther Tool for Language Recognition (ANTLR). ANTLR provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions [15]. Figure 3.1 illustrates the data flow when parsing source code of a programming language starting with an input string.

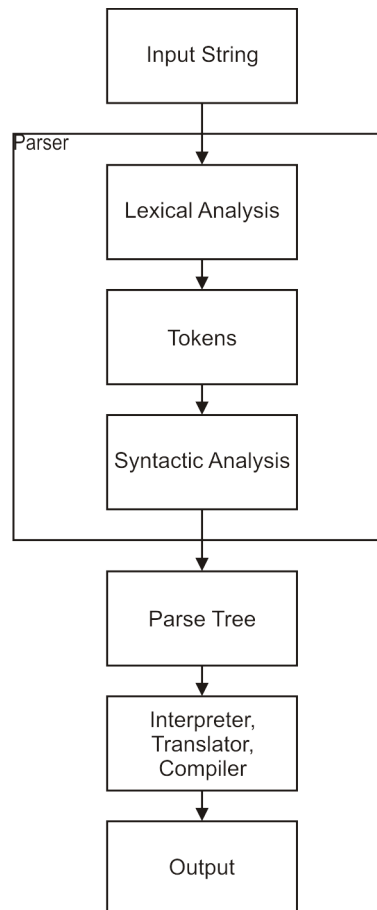


Figure 3.1: The image shows the data flow when parsing source code of a computer programming language. Starting with an input string the parser performs a lexical analysis in order to get tokens. In the next step a syntactical analysis processes the tokens to determine their grammatical structure. A parse tree is generated as a result and serves as data structure for interpreters, translators or compilers.

A collection of descriptions, called grammar, is needed to determine whether input sentences conform to that language, or not. Sentences in terms of language recognition are similar to sentences in a natural language.

It is an expression following certain grammatical and lexical rules forming a meaningful command. The most basic part of a sentence is a character like “a” or “b”. In a natural language a sequence of characters is called a word, whereas in terms of language recognition it is called a token. Starting with a sequence of characters on an input stream a lexical analysis is performed to generate tokens. In a next step, these tokens are analyzed to determine their grammatical structure. This process, called parsing, involves the grammar mentioned above. A convenient way of creating such grammars is to use the graphical development environment ANTLRWorks, which is a tool for developing and debugging such grammars.

Starting with the most abstract concepts of a language down to the elemental parts, one recognizes sentences of a language by defining their implicit tree structure using a grammar. Abstract concepts define the root of the tree and elemental parts represent the leaves. In order to simplify the generation of a parser it is necessary to allow a natural description of the language. However, there are a number of things to consider:

- Allowing a large number of grammars usually leads to parsers that are less efficient and difficult to understand [18].
- It is possible to generate grammar rules that are ambiguous for certain input. This happens when input can be matched by more than one rule defined in the grammar.

Usually the parsing strategy provides a lookahead of finite length in order to identify the correct path through the grammar. Such a recognizer is called *top-down* or *LL* because it recognizes the input from **L**eft to **R**ight, and constructs a **L**eftmost derivation of the sentence. ANTLR introduces a strategy called *LL(*) parsing* which extends the *LL(k) parsing strategy* with lookahead of arbitrary length without explicitly specifying it.

ANTLR is not only capable of creating recognizers which decide whether a sentence is correct or not, but it is capable of creating parsers. A parser checks a sentence for correct syntax and generates a parse tree. For that purpose actions written in the target language can be embedded in the grammar which are then passed over to the resulting translator or interpreter. These actions can be embedded almost anywhere in the grammar and contain class member variables and methods as well as statements executed depending on the recognized input symbols. ANTLR generates recognizers and parsers in a specified target language from a collection of methods derived from grammar rules. As shown in Listing 3.1 these rules contain a description of what to do depending on what they see on the input stream. Possible actions are either matching a symbol or invoking other rules. The grammar in Listing 3.1 matches simple assignment statements for integer values. Figure 3.2 shows the parse tree of the assignment statement “count := 34;”. One can see the tree structure resulting from the nested rules of the grammar.

```

1 grammar statement;
2
3 statement
4   : statementAssignment
5   ;
6
7 statementAssignment
8   : variable ':=' INT ';'
9   ;
10
11 variable
12  : IDENT
13  ;
14
15 INT
16  : '0'..'9'+
17  ;
18
19 IDENT
20  : ('a'..'z'|'A'..'Z')+
21  ;

```

Listing 3.1: This listing shows a grammar defining a simple assignment statement for integer values. Starting with more abstract rules like *statement* defining the root of the tree structure the grammar follows the top-down approach. Elemental rules represent identifiers as well as integer values with their respective rules *IDENT* and *INT*.

Usually the process of translating a language is not accomplished in a single pass of parsing the input sentences. Variable references for example may require a first pass of collecting the definitions and a second pass of resolving the references. An appropriate intermediate representation enables the translator to efficiently execute multiple passes of parsing. For that purpose ANTLR allows the creation of abstract syntax trees (AST). An AST represents input structure in a compact tree form containing only the information needed for further processing. Using a tree walker it is possible to extract information from a tree as well as alter the tree if needed for the next pass of processing. The last step usually emits output considering all the information previously gathered.

For the purpose of creating a suitable programming language meeting the requirements mentioned at the beginning of this chapter it is necessary to create a translator. Going through each grammatical rule of the translator will not benefit the understanding of the language as much as an introduction. Although a complete listing of all grammar rules can be found in the addendum. The resulting programming language is called *Euclides*, following the origin of the name of the *Pascal* programming language to which it is related. The next chapter features an introduction of the language constructs.

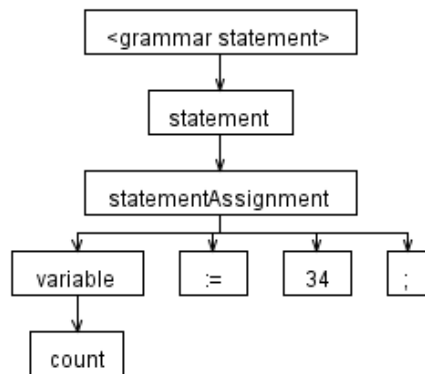


Figure 3.2: The image shows the parse tree of the statement: “count := 34;” obtained using the grammar of Listing 3.1. One can see that the parse tree represents the tree structure of the grammar.

3.1 Euclides

The *Hello World* example shown in Listing 3.2 is the entry point for all programming language introductions. I do not wish to break with that practice. As one can see a program starts with the keyword *program* followed by its name. The main statement block of a program is delimited by the keywords *begin* and *end*. The full stop after the keyword *end* defines the end of the program. To bring the *Hello World* string on the screen the *write* command is used. It takes a string as argument and prints it to the standard output. The *write* command itself is defined within a library called *io*. Libraries in *Euclides* are implemented in so called units in order to be included in a program. As can be seen in line 3 of the example, the keyword *uses* followed by the unit’s name includes it in the program. Units can be listed comma separated after the keyword *uses* in case more than one needs to be included. Two things are noticeable when looking at the example:

- A string is always delimited by single quotes.
- The semi-colon at the end of a line represents a statement delimiter.

The use of comments is shown in Listing 3.3. *Euclides* supports two types of comments:

- A long comment is delimited by curly brackets and can span over several lines. It can be placed anywhere in the source code.

- The tilde symbol introduces short, single line comments. It represents a rotated half of a curly bracket and therefore suggests short comments.

```
1 program hello;  
2  
3 uses io;  
4  
5 begin  
6   write('Hello world');  
7 end.
```

Listing 3.2: This listing shows a *Hello World* example program. The string *Hello World* is printed to standard output using the a unit called *io*. The unit itself is included using the keyword *uses*.

```
1 program comments;  
2  
3 {This is a long and of course very useful comment  
4  and it can span over more than one line.}  
5  
6 begin  
7   ~This is a rather short comment.  
8 end.
```

Listing 3.3: This listing demonstrates the use of the two types of comments in an example program. Comments that possibly span over more than one line are delimited by the curly brackets whereas short comments are introduced by the tilde symbol.

In *Euclides* one must always declare a variable before using it. This is done in a special section introduced by the keyword *var*. A variable cannot be declared outside a *var* section. Each variable declaration consists of an identifier followed by a colon and a type. The following variable types are predefined:

- boolean: can either be true or false
- integer: represents positive and negative integer numbers including zero
- real: represents positive and negative real numbers including zero
- vector: represents a vector holding integer or real values
- matrix: represents a vector of vectors

- string: represents a sequence of characters
- reference: refers to an object
- void: represents a unit type that allows only one value and thus holds no information
- error: represents an *Euclides* error
- array: holds items of any unique variable type available in *Euclides*
- set: represents a collection of items of any unique variable type available in *Euclides*
- record: aggregates several items of possibly different variable types available in *Euclides*
- user-defined: represents one of the above mentioned types

In *Euclides* it is possible to declare user-defined types. This is done in a special section introduced by the keyword *type*. Once declared it can be used like a built-in type with all its properties. Each type declaration consists of an identifier followed by a colon and a type. Constants are defined in a special section introduced by the keyword *const*. Only a small set of types can be used for constants: string, matrix, vector, real, integer and boolean. Because of the small set of possible types, there is no need to declare a constant. The type of the constant can be uniquely identified with the value. As with variables, types and constants can only be declared or defined in their special sections. Listing 3.4 shows declaration and definition of variables as well as the use of constants and types. A few things are noticeable when looking at the example:

- A *var*, *const* or *type* block precedes a main statement block.
- The operators for variable declaration and definition are distinct to eliminate obscurities. The operator for variable declaration is (':') whereas the operator for variable definition is (':=').
- A *var*, *const* or *type* block has no delimiting *begin* or *end* because it has a well-defined and distinguishable position in a *Euclides* program and therefore needs no delimiters.

Variables and constants can be connected with operators. *Euclides* supports the following operators:

- 'and': infix operator, represents a logical and
- ':=': infix operator, represents an assignment operator

- '=': infix operator, tests the equivalence of two values
- '>=': infix operator, tests if the value of the left expression is greater than or equal to that of the right
- '>': infix operator, tests if the value of the left expression is greater than that of the right
- 'in': infix operator, tests if the value of the left expression is in that of the right
- '<=': infix operator, tests if the value of the left expression is less than or equal to that of the right
- '<': infix operator, tests if the value of the left expression is less than that of the right
- '-': prefix or infix operator, represents a sign or a subtraction
- 'not': prefix operator, represents a logical not
- '!=': infix operator, tests the negated equivalence of two values
- 'or': infix operator, represents a logical or
- '+': prefix or infix operator, represents a sign or an addition
- '#': prefix operator, represents a sizeof operator
- '/': infix operator, represents a division
- '*': infix operator, represents a multiplication

In contrast to *Pascal* there is no distinction between procedures not returning a value and functions returning a value in *Euclides*, methods always return a value. The definition of methods resides outside the main statement block of a program like shown in Listing 3.5. A method can have their own blocks for variables, types and constants. The return value type of a method is specified after the end of the parameter list. By definition a return value gets named after the method itself. The syntax of some of the control structures supported by *Euclides* is demonstrated in the method's body.

```
1 program variables ;
2
3 const
4   c := 'constant' ;
5
6 type
7   i : integer ;
8
9 var
10  v : i ;
11  s : string ;
12  b : boolean ;
13
14 begin
15   v := -3 ;
16   s := c ;
17   b := true ;
18 end .
```

Listing 3.4: This listing demonstrates the use of variables, constants and types. Constants are created within a *const* block as can be seen in line 3 and 4. Variables and types are defined in a *var* respectively *type* block. It is important to note that constants, variables and types can only be created respectively defined in their particular blocks.

Another important construct is the *unit*. A *unit* represents a collection of methods which can be utilized in programs. In other programming languages it is often referred to as *library*. Similar to a program, the keyword *unit* followed by an identifier introduces a *unit*. In order to be able to use a *unit* in a program it is necessary to include it. This is done by using the keyword *uses* followed by the name of the unit.

```

1 program bubblesort;
2
3 type
4   int_array : array of integer;
5
6 var
7   a : int_array;
8
9 method bubblesort(numbers : array of integer) : int_array;
10 var
11   i, j, temp : integer;
12
13 begin
14   bubblesort := numbers;
15   for i := (#(numbers)-1) downto 1 do
16     begin
17       for j := 2 to i do
18         begin
19           if (bubblesort[j-1] > bubblesort[j]) then
20             begin
21               temp := bubblesort[j-1];
22               bubblesort[j-1] := bubblesort[j];
23               bubblesort[j] := temp;
24             end
25           end
26         end
27       end
28     end
29   begin
30     a[1] := 23;
31     a[2] := 45;
32     a[3] := 34;
33
34     bubblesort(a);
35   end.

```

Listing 3.5: This listing shows the use of methods by implementing the bubble sort algorithm. The algorithm itself is implemented in the method `bubblesort`. In the main block of the program example values are created and the `bubblesort` method is called.

3.2 Parsing Euclides

As depicted earlier, a grammar describes the syntax of a language using a set of rules. In order to match input to a set of rules it is necessary to group the input into tokens and consecutively to sentences. A lexer reads characters from the input stream and groups them into tokens. On this rather low level it is possible to discover lexical errors, such as erroneous characters. Listing 3.6 shows a rule defining possible characters of an identifier. An error is emitted when the input character does not match the rule. The rule itself is grouped into 2 subrules delimited by brackets. The first subrule matches the first character which can be any letter from a to z including upper case letters. Alternatives are separated by the pipe character describing a disjunction. A modifier at the end of line 2 enables the second subrule to be matched 0 or more times ensuring a variable length of an identifier. In contrast to the first subrule, additional characters like numbers or underlines are allowed.

```
1 IDENT
2 : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'*)
3 ;
```

Listing 3.6: A rule that defines the characters allowed for an identifier. The first character can be any letter from a to z including upper case letters. Any other following character can additionally be a number or an underline.

Input sentences are matched by more abstract rules and errors emitted when no viable alternatives are found. During parsing, an AST is generated according to the matched input. Listing 3.7 shows a rule that matches program or unit heading. Several things are noticeable in this example:

- A rule can also have one or more alternatives introduced by a pipe character (line 5).
- The call sign modifier ('!') omits the token in the AST.
- The circumflex modifier ('^') sets the token as AST root node.
- Anything that needs to be done beyond recognizing the syntax of *Euclides* is performed by so called actions. Embedded actions are delimited by curly brackets and can be placed anywhere in a rule. They are written in the target language and usually refer to token attributes.

```

1 programHeading
2   : PROGRAM^ identifier {
3     environment.currentScope.setScope(NamingStandard.
4       getProgramString($identifier.text));
5   } SEMI!
6   | UNIT^ unitidentifier {
7     environment.currentScope.setScope(NamingStandard.getUnitString(
8       $unitidentifier.text));
9   } SEMI!
10  ;

```

Listing 3.7: Rule *programHeading* matches either the heading of a program or a unit. The embedded Java code sets the root scope for both alternatives.

Often it is necessary to pass information from elemental rules to more abstract rules. Therefore tokens have a set of attributes like the text matched for the token or the line number in which the token occurs. ANTLR also allows arbitrary return values for rules when the token attributes do not suffice. Listing 3.8 shows the rule matching a constant value of type string. It returns value and type to the parent rule(s) in the hierarchy. One can see that the return value type is a user-defined class whereas the returned string is obtained through a token attribute.

```

1 constantString returns [String value, Type type]
2   : STRING_LITERAL
3   { $value = $STRING_LITERAL.getText();
4     $type = new StringType(); }
5   ;

```

Listing 3.8: Rule *constantString* returns value and type of the string constant to the parent rule(s) in the hierarchy.

Another important aspect is the collection of information needed for further processing. This is done using actions. The following information is collected during parsing:

- Symbols are collected and stored in a data structure called symbol table. In this context a symbol is either a constant, a variable, a type or a method. To simplify the validation process, references to symbols are also stored in the AST. In order to be able to store references in the AST the standard tree node *CommonTree* is extended to fit the needs of the task.
- The scope of a token is also determined and stored in the appropriate node of the tree. It represents the visibility of the token and is used in the validation process. The idea behind scoping is the keep variables

and methods in different parts of a program distinct from one another. In *Euclides* it is possible to define and use two variables with the same name in different methods, as long as the methods are in the same scope. Beginning with the root scope, every method definition represents a new scope. The scope of the new method contains all symbols defined in the surrounding scope but no symbol of the new scope can be referenced outside.

The naming standard of symbols collected in the symbol table is an important aspect. It is used to identify the scope of a symbol. First of all it is crucial to know that all characters defined in the naming standard are in upper case. In order to be uniquely identifiable, all other strings, like names defined in a *Euclides* program, are converted to lower case. The following list describes the naming standard for each symbol:

- A program string gets the prefix “P”.
- A unit string gets the prefix “U”.
- A method string gets the prefix “M” and the list of parameters as suffix.
- A variable string gets the prefix “V”.
- A constant string gets the prefix “C”.
- A type string gets the prefix “T”.

The list of parameters when creating a method string is delimited by the characters “X” and “Y”. In case more than one parameter is defined, they are separated using the “Z” character. All implemented types as well as the user-defined type have their own representations in the naming standard:

- boolean: “EBN”
- integer: “EIR”
- real: “ERL”
- vector: “EVR”
- matrix: “EMX”
- string: “ESG”
- reference: “ERE”
- void: “EVD”

- error: “EER”
- array: “SAR”
- set: “SST”
- record: “SRD”
- user-defined: “EUF”

The leading characters “E” and “S” stand for elementary respectively structured. For example, when creating a variable *a* in a var block inside the method *foo* (with no parameters) of the program *test* it will get the name *PtestMfooXYVa*. Given the name alone it is therefore possible to identify the variable’s location. Furthermore, the *Euclides* source code file extension “ecs” as well as the *Euclides* unit file extension “ecu” are also defined in the naming standard.

When collecting symbols the following information is stored for the different symbols:

- A constant consists of a name, a trace indicating its scope, a reference to a type, a value and references to its definition as well as its occurrences in the source code.
- A variable consists of a name, a trace, a reference to a type and references to its definition as well as its occurrences in the source code. In contrast to a constant there is no need to store a value.
- A data type consists of a name, a trace, a reference to a type and references to its definition as well as its occurrences in the source code.
- A method consists of a name, a trace, references to its parameters and the return value, references to its definition as well as its occurrences in the source code and references to all symbols defined within the method.

In Listing 3.9 a rule matching a constant definition is shown. The action block inside the rule handles the following tasks:

- Creating a new constant holding name, scope, type and value.
- Setting a reference of the constant in the AST as well as a reference of the AST in the constant.
- Adding the constant to the symbol table.
- If defined inside a method, the constant is added to the method as well.

```

1 constantDefinition
2   : identifier EQUAL^ c=constant {
3     Constant tmp = new Constant(NamingStandard.normalize(
4       $identifier.text), environment.currentScope.getScope() +
5       NamingStandard.getConstantString($identifier.text), $c.type
6       , $c.value);
7     environment.symtab.addConstant(tmp);
8     crossReference(tmp, (EuclidesAST)$identifier.tree);
9     // if defined inside a method, then add to appropriate method
10    if (environment.currentScope != environment.globalScope) {
11      Method m = (Method)environment.symtab.getMethodByTrace(
12        environment.currentScope.getScope());
13      m.addConstant(tmp);
14    }
15  }
16 ;

```

Listing 3.9: Rule *constantDefinition* matches a constant definition and stores name, scope, type and value in a symbol table. It also stores a reference of the constant in the AST and a reference of the AST node in the constant for easy access. When the definition of a constant occurs inside a method, the constant is added to the appropriate method.

Similar actions handle the storage of information for variables, types, and methods.

When deciding whether to include a token in the AST or not it is often necessary to create a certain hierarchical structure for a tree walker to work efficiently. The usage of modifiers to obtain a suitable hierarchical structure does not always work out. Therefore tree rewrite syntax is used in order to have more control over the structure. In Listing 3.10 the imaginary token *METHOD_CALL* is introduced as subtree root node. Imaginary tokens are not associated with input characters and are specified in a special section in the grammar. When using tree rewrite syntax no modifiers are allowed to be set in the rule itself. Instead, there is a section after the rule introduced by the arrow operator (*'->'*) that allows modification of the tree structure. A tree walker can now easily detect a method call by evaluating the first node of the subtree.

The AST generated by the parser is used in the validation process described in the following chapter.

3.3 Validating syntax and semantics

While a grammar checks input for syntactical correctness, there are no control constructs for semantic correctness in ANTLR. Before transpiling source

```

1 statementMethodCall
2   : identifier LPAREN (expression (COMMA expression)*)? RPAREN ->
3     ^(METHOD_CALL identifier expression*)
  ;

```

Listing 3.10: Rule *statementMethodCall* shows the use of rewrite syntax to change the structure of the AST. The imaginary token *METHOD_CALL* is introduced as a subtree root node to simplify the detection of a method call.

code it is needed to perform as many checks as possible to make sure the next instance gets widely error less input. These checks have to be performed relying on actions that acquire information. However, this information has to be validated in a second step. Therefore the generated AST is processed using a tree walker. Missing information is acquired using the AST as well as the generated data structures. A validation process checks the input for semantic correctness.

Validating Euclides

Input is usually generated in the form of source code. Before starting to validate source code, it is necessary to collect information from all included files. For example this is crucial when referencing symbols defined in units. In case a unit is not parsed in before a referencing source code file is validated, it will certainly result in errors. It is important to note that information from different files is stored in one instance of the symbol table.

In advance of validating *Euclides* source code it is necessary to check the symbol table for missing information. For example it is possible to define a type in a unit and then use it in another unit or in the program file. At the time the program file is parsed in, the information regarding the user-defined type is yet to be parsed in. At this point it is not possible to set the reference to the user-defined type. Therefore it is imperative to check the symbol table for missing references and set them. Similar reasons lead to the emergence of duplicate symbols. Methods, types, constants and variables are therefore checked for duplicates in advance.

Euclides source code is validated using the class *Validator*. The AST generated by the parser, as well as a reference to the symbol table are handed over to the class as arguments. The entry point of the class is a tree walker that serves as a dispatcher for the following validation tasks:

- Method calls
- Variable assignments
- Variable definitions

- Type definitions
- If statements
- While statements
- Case statements
- For statements
- Repeat statements
- Try, catch statements

Whenever the tree walker recognizes a token that introduces one of the language constructs mentioned above, it invokes the appropriate validation method. An AST subtree representing the entry point for the validation process is handed over to every method. The methods itself evaluate every node of the subtree and report possible errors using a *ValidationException* which includes filename, line, position, error code and error text. The validation process is then stopped and the error is displayed.

Method calls

A method call like shown in Listing 3.11 line 12 can be rather complex to validate. In the example only 2 parameters of type real are passed over as arguments, but every possible combination and type representing a parameter has to be considered. One can think of a return value of a method call, or an element of an array.

```
1 program methodCall;
2
3 var
4   result : real;
5
6 method pow(a, b : real) : real;
7 begin
8   pow := a + b;
9 end
10
11 begin
12   result := pow(2.0, 2.0);
13 end.
```

Listing 3.11: A small program showing the definition of a method as well as a method call.

The first step is a loop over the child nodes to validate the parameters which can be of the following types:

- A variable.
- A constant.
- An anonymous type, e.g. an integer number.
- The return value of another method call.
- An operator combining any of the types mentioned above.

In case the child node is of the type identifier, a lookup in the symbol table is invoked to get the appropriate symbol which can be a variable or a constant. When a variable or a constant is found, the type and the scope of the symbol are stored for further processing. However, records and arrays have to be handled in a different way. They are identified by either a square bracket in case of an array or a dot in case of a record. In order to be validated they are dereferenced using their own methods. Whenever a symbol is used in the source code, a reference of the tree node representing the symbol is stored in the symbol itself to be able to track the occurrences. Another possible parameter can be an anonymous type. They are treated like symbols only they are much easier to handle. Finding type and scope is a trivial task. In case a parameter is another method call it is handled via a recursive call of the validating method itself. The return value of the method call represents the parameter. When an operator is identified as parameter it has to be resolved down to the resulting data type in order to be evaluated in the next step.

The last step is to find a method definition that matches with the collected information. Only now that all parameter types are known it is possible to search for a matching method. This is necessary because methods can differ only in their parameters and therefore all parameter types have to be known in order to find the correct method. Finally scope checks ensure that the method call as well as all parameters are visible.

Variable assignments

The first step when validating a variable assignment is a lookup in the symbol table to get the appropriate left-hand variable. The type of the variable is stored as a target type to be used to match the right-hand variable type later on. In case the left-hand variable type is either a record or an array, they have to be validated and dereferenced first. A loop over the child nodes validates the right-hand types which can be:

- A variable.
- A constant.
- An anonymous type, e.g. an integer number.
- The return value of a method call.
- An operator combining any of the types mentioned above.

The checks performed for the appropriate right-hand types are similar to the checks for the method parameters mentioned earlier. The only difference is that the right-hand type is evaluated directly because only one variable can be assigned a value at a time. When the type of the right-hand expression is determined a lookup whether an assignment operator is defined for the two types or not is performed.

Variable definitions

As mentioned in section 3.2, a variable has to be defined in a variable block. A subtree representing a variable block is handed over to this method. In a loop, every variable definition in a variable block is validated. The first step is a lookup in the symbol table to get the variable. In case the type of the variable is user-defined, e.g. a record type, a lookup in the symbol table clarifies whether this type exists or not. Finally, the scope of the variable is matched against the type.

Type definitions

Similar to the checks performed for variable definitions, every type definition in a type block is validated. Therefore a subtree representing a type block is handed over to the method. In case a user-defined type is found, a lookup in the symbol table is issued to check whether this type exists or not. A final check matches the scope of the types.

If statements

As shown in Listing 3.12 an if statement begins with the keyword *if* followed by the condition in brackets. The keyword *then* introduces the first branch of the condition, whereas *else* introduces the optional second branch. In case several statements are needed in a branch the keywords *begin* and *end* are used as delimiters.

The condition of the statement should be of type boolean, therefore the target type is set to boolean. The following statements can be of type boolean and are validated:

- A variable.
- A constant.
- The return value of a method call.
- An operator combining any of the types mentioned above.

```

1 if (a > b)
2 then
3   a := max;
4 else
5   begin
6     ~just a comment
7     b := max;
8   end

```

Listing 3.12: This fragment of code represents an if statement. It is introduced by the keyword *if* followed by the condition and the keyword *then*. The branch introduced by the keyword *else* is optional.

When the type is determined a lookup whether an equal operator is defined for the two types or not is performed.

While statements

A while statement begins with the keyword *while* followed by the condition in brackets. The keyword *do* introduces the statement block. In case several statements are needed in the block the keywords *begin* and *end* are used as delimiters. The while statement is handled very similar to an if statement because of the boolean condition. Therefore it needs no extra explanation.

Case statements

A case statement like shown in Listing 3.13 begins with the keyword *case* followed by the statement to be switched. The different branches are introduced by the values of the statement and usually enclosed by the keywords *begin* and *end*.

For the validation of a case statement it is necessary to determine the type of the case. Only variables, method calls and operators represent switchable cases. Once the type of the case is validated, it is set as target type. A loop over all branches determines whether an equal operator is defined for the target type and the actual branch type, or not. Possible types for branches can be any of the implemented data types or return values of method calls.


```
1 case number of
2   1 : begin
3     ~code
4   end
5   2 : begin
6     ~code
7   end
8 end
```

Listing 3.13: This listing shows a fragment of code representing a case statement. It is introduced by the keyword *case* followed by the statement to switch through and the keyword *of*. Subsequently the different cases are defined. An optional else branch represents the default case.

For statements

Listing 3.14 shows a for statement beginning with the keyword *for* followed by the iteration of the loop variable.

```
1 for i := 1 to 3 do
2 begin
3   element[i] := 0;
4 end
```

Listing 3.14: This fragment of code represents a for statement. It is introduced by the keyword *for* followed by the numeric ranges and the keyword *do*. A for statement can either go from a starting value *to* an end value, or *downto*.

The loop variable of a for statement must be declared in a *var* block like every other variable that is used in a program. Therefore it should be in the symbol table which is validated with a lookup. Once the variable is obtained a match against the scope of the for statement is conducted. A loop variable can either be of type integer or type real. That applies to the numerical ranges as well. With the target type set to either integer or real it is determined whether an equal operator exists for the given numerical limits, or not. Besides the standard types real and integer the limits can be:

- A variable.
- An element of an array.
- A part of a record.
- The return value of a method call.

Repeat statements

Listing 3.15 shows a repeat statement which is introduced by the keyword *repeat* followed by the code to be evaluated an arbitrary number of times. The conditional expression at the end of the statement has to be of type boolean.

```
1 repeat
2   begin
3     count := count + 1;
4   end
5 until
6   count <= max;
```

Listing 3.15: This listing shows a fragment of code representing a repeat statement. It is introduced by the keyword *repeat* followed by the statements to execute *until* a certain condition is met.

Therefore the target type of a repeat statement is set to boolean. For the conditional expression the following types are allowed:

- A variable.
- A constant.
- An element of an array.
- A part of a record.
- The return value of a method call.
- A boolean type.

When the type is determined a lookup whether an equal operator is defined for the two types or not is performed.

Try, catch statements

Listing 3.16 shows the use of try, catch statements. A try, catch statement is for exception handling similar to Java. The target type is set to error and represents the only statement that needs to be validated. Types like variables, an element of an array, a part of a record or return values of method calls can be of type error. After determining the type it is matched against the target type.

```
1 try
2   begin
3     read(input);
4   end
5 catch (err)
6   begin
7     write('error reading input ');
8   end
9 end
```

Listing 3.16: This fragment of code shows a try, catch statement. It is introduced by the keyword *try* followed by the statements that may produce an error. The error is handled in the *catch* part of the statement.

3.4 Transpiling Euclides

The generation of output is the last step in the processing pipeline. Output is generated using transpilers which translate *Euclides* source code into source code of another programming language, e.g. Java or even *Euclides* itself. A symbol table along with a collection of ASTs represents the input of a transpiler. In order to generate output the ASTs have to be processed using tree walkers and information has to be handed over to a template engine. The template engine StringTemplate [16] is used to generate source code due to the fact that ANTLR uses this template engine and that it is available as a separate product. Another mentionable property is that it strictly enforces model-view separation. The *Hello World* example in Listing 3.17 shows the use of StringTemplate. A new template *hello* is generated in line 6. The delimiter “\$” is used to specify an attribute called *name*. In the next line the string “World” is assigned to the attribute *name* using the *setAttribute* method. The output of the program is “Hello, World”.

Two things are noticeable when working with StringTemplate:

- It is possible to nest templates to simplify the handling of possibly complicated strings.
- Templates can be defined in a file and are therefore separated from the source code.

Three different transpilers are implemented to be used as views for the modeling process on the one hand and to actually produce executable code on the other hand:

- *Euclides* transpiler: generates *Euclides* code
- HyperText Markup Language (HTML) transpiler: generates a HTML page displaying *Euclides* code

- Java transpiler: generates executable Java code

```
1 import org.antlr.stringtemplate.*;
2 import org.antlr.stringtemplate.language.*;
3
4 public class HelloWorld {
5     public static void main(String[] args) {
6         StringTemplate hello = new StringTemplate("Hello, $name$",
7             DefaultTemplateLexer.class);
8         hello.setAttribute("name", "World");
9         System.out.println(hello.toString());
10    }
```

Listing 3.17: This listing shows a *Hello World* example using `StringTemplate`. The template *hello* is defined in line 6. It requires a parameter called *name*. In the following lines the parameter is set and the template is printed to standard output.

In the following chapter the *Euclides* transpiler used for source code refactoring is described.

Euclides Transpiler

The *Euclides* transpiler generates *Euclides* code directly from a collection of ASTs. In this case no symbol table is needed because all the required information is contained in the ASTs. The generation of *Euclides* code is an important part of the modeling process. Every time a change in the source code happens, it is necessary to rerun the parser and the validator to alert possible errors. The source code as well as the internal data structures have to be consistent at any time. As mentioned earlier an AST is generated during parsing and is highly interlinked with the input stream. Nodes of an AST reference the token stream while the token stream itself references the character stream. Therefore it is not easily possible to change the AST's structure and then generate new *Euclides* code out of it. It is necessary to incorporate changes in the source code during the generation of the *Euclides* code and subsequently generate new ASTs and new source code.

The next important detail are comments in the source code. Usually comments can be disregarded when generating executable code, but for the purpose of generating *Euclides* code they must be preserved. Therefore the *CommonTree* node is extended by another reference that stores all hidden tokens including comments. Because of the fact that hidden tokens are not included in the AST it is essential to set them in a node that is included, otherwise they are lost. By definition this is the next node that was created

out of a non hidden token. Several tokens are declared hidden during the lexical analysis:

- native code: is inserted when emitting output
- annotations: are inserted when emitting output
- whitespaces: are disregarded for the moment
- comments: are inserted when emitting output

The task of walking a tree and generating output is handled by a class called *EuclidesWriter*. A node introducing a statement is evaluated by its own method. In such a method the whole subtree of the statement is processed and the resulting output is then passed on to next method in the tree hierarchy as argument to be included in the evaluation. A `StringTemplate` for every statement handles the generation of *Euclides* code. This means that every template is somehow contained within the outermost template called *program* to reflect the tree structure of the AST. Accordingly the *program* template consists of templates for program heading, constant definitions, variable definitions and so on. All these templates are collected in a single `StringTemplate` group file. Listing 3.18 shows parts of the group file for *Euclides* output.

```
1 group EuclidesWriter ;
2
3 program(item) ::= "<item>."
4
5 programHeading(x) ::= "program <x>;<\n>"
6
7 unitHeading(x) ::= "unit <x>;<\n>"
8
9 usesPart(x) ::= "uses <x>; separator=\" , \";<\n>"
10
11 constantDefinition(statement) ::= <<const
12   <statement>
13 >>
14
15 statementBlockBegin(item) ::= <<begin
16   <item>end
17 >>
```

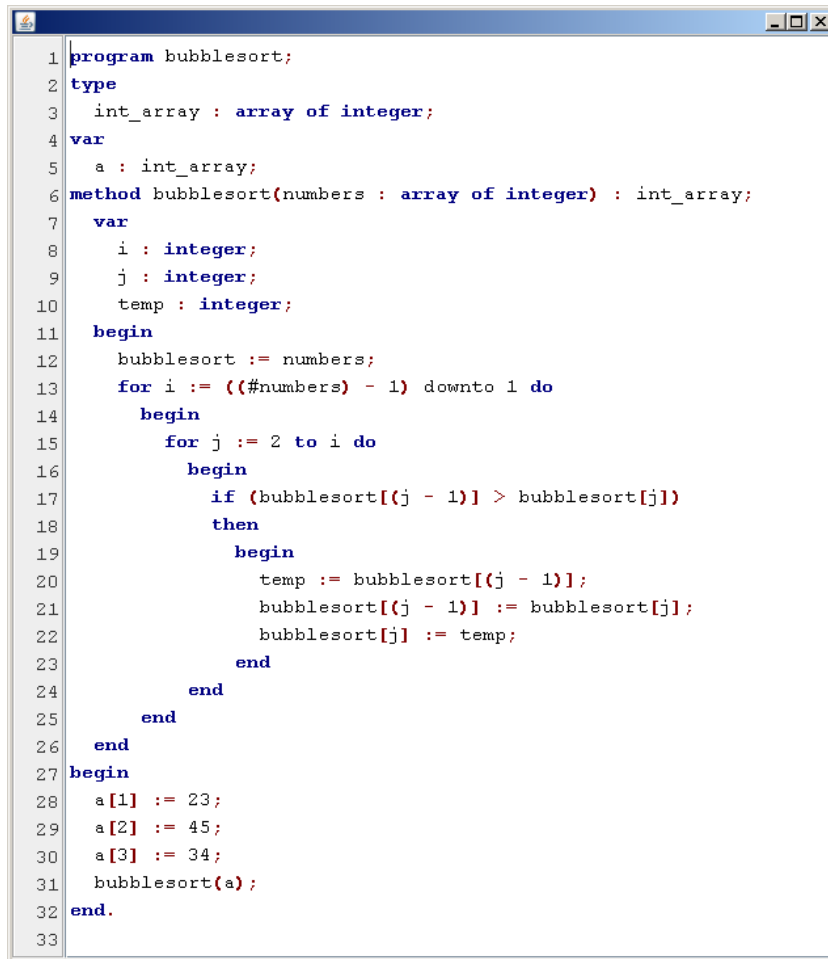
Listing 3.18: This listing shows parts of the `StringTemplate` group file for *Euclides* output. One can see the definition of templates representing the structure of the language. In order to create meaningful output the templates are called nested.

One can see that a template definition consists of a template name followed by attributes in brackets. Single line templates are delimited by double

quotes, whereas multi-line templates are delimited by double angle brackets. Within a template, attributes can be referenced by their names delimited by angle brackets. It is interesting to note that multi-valued attributes are possible. This means that one can set the same attribute multiple times. When working with multi-valued attributes it may become necessary to separate values. This can be achieved by using a separator that can be customized.

Before the output of a `StringTemplate` is passed on to the next one in the hierarchy it is checked whether there are hidden tokens to be considered or not. In case there are, they are inserted in front of the evaluated statement.

In order to change a symbols name, or reset a constants value a data structure called *AlteredSymbols* needs to be filled with the changes. During the process of generating output this data structure is pulled up to incorporate possible changes. Once these changes are taken into account, re-evaluation of the generated source code leads to stable results. Figure 3.3 shows the output of a generated *Euclides* file using a small text editor with syntax highlighting.



```
1 program bubblesort;
2 type
3   int_array : array of integer;
4 var
5   a : int_array;
6 method bubblesort(numbers : array of integer) : int_array;
7   var
8     i : integer;
9     j : integer;
10    temp : integer;
11  begin
12    bubblesort := numbers;
13    for i := ((#numbers) - 1) downto 1 do
14      begin
15        for j := 2 to i do
16          begin
17            if (bubblesort[(j - 1)] > bubblesort[j])
18              then
19                begin
20                  temp := bubblesort[(j - 1)];
21                  bubblesort[(j - 1)] := bubblesort[j];
22                  bubblesort[j] := temp;
23                end
24              end
25          end
26        end
27      begin
28        a[1] := 23;
29        a[2] := 45;
30        a[3] := 34;
31        bubblesort(a);
32      end.
33
```

Figure 3.3: This figure shows a small text editor with syntax highlighting displaying the *Euclides* bubble sort example file generated out of its AST.

Chapter 4

Views

Displaying source code alone often does not contribute to a better understanding. Therefore different views of the source code are necessary. As a first view an HTML version of the source code is presented in order to allow source code analysis. It makes use of hyper links in order to simplify the navigation in the source code. Additionally it introduces an overview section displaying a list of important symbols. In order to enable source code interpretation and inspection, a Java transpiler serves as a second view. For that purpose the *Euclides* source code is transpiled to Java code. A runtime environment developed for that purpose provides the corresponding data types and operators. It also allows for a framework to control the execution of transpiled programs for example to implement a debugger. The two different views of the source code are introduced in this chapter.

4.1 HTML Transpiler

The HTML transpiler generates *Euclides* code embedded in a HTML page. Similar to the *Euclides* transpiler, the code is directly generated from ASTs. Additionally a collection of all variables and constants is also generated to provide an overview. This overview is generated using the symbol table.

The generation of the source code is handled by the same templates used for the *Euclides* transpiler. To benefit from the possibilities that arise when using HTML code, symbols in the *Euclides* code link to the equivalent symbols in the overview. Therefore all symbols in the overview are extended with anchors. In order to be able to insert HTML code in the output it is necessary to exchange templates used to generate *Euclides* output. Listing 4.1 shows the `StringTemplate linkitem` that is called whenever a variable or constant occurs in the *Euclides* code. One can see the link to the appropriate anchor that is defined using the templates *constantAssign* and *variableColon*.

Another important aspect when looking at source code is to be able to

```

1 constantAssign(item) ::= <<\<a name="<first(item)>"\><first(item)
   >\</a\> := <last(item)>
2 >>
3
4 variableColon(item) ::= <<\<a name="<first(item)>"\><first(item)
   >\</a\> : <last(item)>
5 >>
6
7 linkitem(item) ::= "\<a href="#"#<item>"\><item>\</a\>"

```

Listing 4.1: This listing shows parts of the StringTemplate group file for HTML output that generates anchors as well as links to anchors.

```

Usage:
  java -jar ECSCompiler.jar infile.ecs [options] outfile.jar

Options:
  -Ipath, /Ipath      add include path
  -Lpath, /Lpath      add library path
  -Tpath, /Tpath      set temporary path
  -Plog, /Plog        set protocol log file

```

Listing 4.2: This listing shows the help text when starting the *ECSCompiler* with the argument “-h”.

determine line numbers. For this purpose the output of the HTML transpiler is extended with line numbers. In order to preserve the layout of the code line numbers are zero padded to be of the same length. Figure 4.1 shows the bubblesort example as a generated HTML file.

4.2 Java Transpiler

Unlike *Euclides* and HTML transpiler, the Java transpiler generates executable code. In fact it generates a jar file. The process of generating Java code from *Euclides* code is rather complex. A class called *ECSCompiler* handles the process. Starting the *ECSCompiler* with the argument “-h” displays the help text as shown in Listing 4.2. One has to define the input file, several options concerning paths as well as the output jar file.

Calling the *ECSCompiler* with meaningful parameters starts the compilation process:

1. The *Euclides* file is parsed in as well as all units associated with it.
2. The resulting symbol table is checked for duplicate symbols and all missing references are set.

```
Variables:
a : userdef type 'int_array' equivalent to 'array with element type 'integer''
i : integer
j : integer
temp : integer

01: program bubblesort;
02: type
03:   int_array : array of integer;
04: var
05:   a : int_array;
06: method bubblesort(numbers : array of integer) : int_array;
07:   var
08:     i : integer;
09:     j : integer;
10:     temp : integer;
11:   begin
12:     bubblesort := numbers;
13:     for i := (#numbers) - 1 downto 1 do
14:       begin
15:         for j := 2 to i do
16:           begin
17:             if (bubblesort[j - 1] > bubblesort[j])
18:               then
19:                 begin
20:                   temp := bubblesort[j - 1];
21:                   bubblesort[j - 1] := bubblesort[j];
22:                   bubblesort[j] := temp;
23:                 end
24:             end
25:           end
26:         end
27:       begin
28:         a[1] := 23;
29:         a[2] := 45;
30:         a[3] := 34;
31:         bubblesort(a);
32:       end.
```

Figure 4.1: This figure shows the bubblesort example as a generated HTML file used for source code analysis.

3. Hidden tokens are acquired and referenced in the appropriate nodes of the AST.
4. All generated ASTs are validated.
5. The Java transpiler is called to generate Java files.
6. The Java files are compiled and the output is packed in a jar file.

The *Euclides* code in Listing 4.3 is a test program used to illustrate the process of creating Java code. It consists of a method *foo* which takes an integer *x* as argument and returns an integer. In the main statement block the method is called with the argument “1” represented by the constant *c*.

The result is then assigned to the variable *v*.

```
1 program test;
2
3 const
4   c := 1;
5
6 var
7   v : integer;
8
9 method foo(x : integer) : integer;
10
11 begin
12   foo := x;
13 end
14
15 begin
16   v := foo(c);
17 end.
```

Listing 4.3: This listing shows an example program to be used to illustrate the Java transpilation process.

The Java transpiler generates four Java files out of a *Euclides* file:

- Main.java
- Constants.java
- Variables.java
- Methods.java

Main.java represents the entry point for a program. Listing 4.4 shows parts of the file representing the entry point of the transpiled *Euclides* program. The first thing to notice are the two member variables *factory* and *controller* which hold a reference to *ECSCompilerFactory* respectively *ECSCompilerController*. The factory is used to create constants, variables and operators, to initialize variables with default values as well as to store information about methods. The idea behind the factory is to make certain operations exchangeable. When the Java transpiler is used in the context of a user interface, it may be necessary to know when an operator is used or a variable is initialized. In case it is used as a stand-alone application there is no need to alert such operations. Basically the same idea can be applied to the controller as well. Its purpose is to allow debugging generated Java code. Therefore it is necessary to know whenever a method is entered or left as well as to set markers in the call stack to allow handling of exceptions. When used as a stand-alone application such operations need not be implemented.

Another important thing is the method *execute()*. The try block is populated with calls to the *execute()* methods of all transpiled *Euclides* source files associated with the program. That means calls to the main statement blocks of the program and all units. In this case it is the call to the *execute()* method of the class *Ptest*.

Every method and main statement block of a *Euclides* source file has its own subclass in the class *Methods*. The classes are located in the file *Methods.java*. Listing 4.5 shows the file generated using the example program mentioned above. One can see that the class *Methods* consists of two subclasses called *MfooXEIRY* and *Ptest*. *MfooXEIRY* is the subclass that represents the *Euclides* method foo, hence the name. Whereas *Ptest* is the subclass that represents the main statement block of the *Euclides* example program. Every subclass consists of three methods:

- *begin()*: This method calls the *enterMethod* method of the controller to indicate that the program now enters this method. In case variables are defined in a var block inside the method, the *enterScope* method is called for each variable to initialize it.
- *end()*: This method calls the *exitMethod* method of the controller to indicate that the program now exits this method. For each variable defined in a var block inside the method, the *exitScope* method is called to indicate that they are no longer needed.
- *execute()*: This method represents the statement block of the method. In case the *Euclides* method has parameters they are passed to this method. After the *begin* method is called the return value is created and initialized. The controller is indicated that this method is ready to be started using the *startMethod* call. Consecutively the transpiled statements of the *Euclides* statement block are inserted. Finally the *end* method is called and the return value is specified in a return statement.

The static part of each subclass contains a call to the factory method *createMethod*. This call provides the method with three parameters. The first one is an instance of *ASTInformation* which will be discussed later. The other two parameters are an array of the parameters of the method respectively its return value.

Variables are declared static in a special class called *Variables*. This class includes variables of all var blocks found in the *Euclides* code. However, parameters and return values are not included. This results in a problem when a method is defined inside a method itself. The inner method should have access to all variables in its scope including the parameters of the outer

```

1 package ecs;
2
3 import cgv.euclides.runtime.Controller;
4 import cgv.euclides.runtime.Factory;
5 import cgv.euclides.runtime.implementation.ECSCompilerController;
6 import cgv.euclides.runtime.implementation.ECSCompilerFactory;
7
8 public class Main {
9     private static Factory factory = new ECSCompilerFactory();
10    private static Controller controller = new ECSCompilerController
11        ();
12    private static String [] arguments = null;
13
14    ...
15
16    public static String [] getArguments() {
17        if (arguments == null)
18            return new String [] {};
19        String [] copy = new String [arguments.length];
20        for (int i = 0; i < arguments.length; i++)
21            copy[i] = arguments[i];
22        return copy;
23    }
24
25    ...
26
27    public static void main(String args []) {
28        arguments = args;
29        execute();
30    }
31
32    private static void execute() {
33        try {
34            Methods.Ptest.execute();
35        } catch (Exception exception) {
36            System.err.println("Runtime Error: " + exception);
37        }
38    }
39
40    ...
41 }

```

Listing 4.4: This listing shows parts of the class *Main* representing the entry point for a program. The method *execute()* is populated with calls to the *execute()* methods of all transpiled *Euclides* source files associated with the example program.

method. Therefore the parameters of the outer method are also passed to the inner method using the *execute* method.

The file *Variables.java* holds the class *Variables*. The sole purpose of this class is to hold variables defined in var blocks of an *Euclides* program. Listing 4.6 shows the class generated for the example program. It holds the variable *Vv*.

The purpose of the class *Constants* shown in Listing 4.7 is similar. The class generated for the example program holds the constant *Cc*.

Transpiling *Euclides* code to Java code is straight forward most of the time. However, certain control structures require special handling. The switch statement introduced in *Euclides* is different from the one used in Java. In contrast to the switch statement in *Euclides*, the one in Java works only with *byte*, *short*, *char* and *int* primitive data types. The *Euclides* version works for all type combinations where an equal operator is defined. In order to implement an equivalent control structure in Java, a template converts the switch statement into a variety of if and else statements. Since the type of the switch statement is not known at that time, control structures using the Java *instanceof* operator are created to resolve the concrete type at runtime of the Java code. The branch with the correct type carries out the switch statement by using a number of if and else statements.

Another difference in control structures arises between *Euclides's repeat until* statement and Java's *do while*. The condition of the Java control structure has to be negated in order to behave equally to *Euclides's repeat until*.

Throughout the listings of this chapter a call to the constructor of the class *ASTInformation* is often found. This class basically holds all information that can be extracted from the AST:

- The symbol's name
- The symbol's scope.
- The name of the program or unit the symbol is defined in.
- The line in the *Euclides* source code.
- The position in the line.
- Comments attached to this symbol.
- Annotations attached to this symbol.
- Other occurrences of this symbol in the *Euclides* code defined as line numbers.

```

1 package ecs;
2 public class Methods {
3     public static class MfooXEIRY {
4         static {
5             Main.getFactory().createMethod(new cgV.euclides.runtime.
6                 ASTInformation(..), new cgV.euclides.runtime.types.
7                 Type[] {Main.getFactory().defaultInteger()}, Main.
8                 getFactory().defaultInteger());
9         }
10    }
11
12    public static void begin() {
13        Main.getController().enterMethod(new cgV.euclides.runtime.
14            ASTInformation(..));
15    }
16
17    public static void end() {
18        Main.getController().exitMethod();
19    }
20
21    public static cgV.euclides.runtime.types.primitive.
22        IntegerType execute(cgV.euclides.runtime.types.primitive.
23            IntegerType Vx) {
24        begin();
25        cgV.euclides.runtime.types.primitive.IntegerType Vfoo =
26            Main.getFactory().createInteger(new cgV.euclides.
27                runtime.ASTInformation(..));
28        Vfoo.setValue(Main.getFactory().defaultInteger().getValue
29            ());
30        Main.getController().startMethod();
31        Main.getFactory().createOperatorAssign().op(Vfoo, Vx);
32        end();
33        return Vfoo;
34    }
35 }
36 public static class Ptest {
37     static {
38         Main.getFactory().createMethod(new cgV.euclides.runtime.
39             ASTInformation(..), new cgV.euclides.runtime.types.
40             Type[] {}, Main.getFactory().defaultVoid());
41     }
42     ...
43 }
44
45 public static void execute() {
46     begin();
47     Main.getController().startMethod();
48     Main.getFactory().createOperatorAssign().op(Variables.Vv,
49         Methods.MfooXEIRY.execute(Constants.Cc));
50     end();
51     return;
52 }
53 }

```

Listing 4.5: This listing shows parts of the class *Methods* representing all method blocks as well as all main statement blocks of the example *Euclides* program.


```

1 package ecs;
2
3 public class Variables {
4     public static cgv.euclides.runtime.types.primitive.IntegerType
        Vv = Main.getFactory().createInteger(new cgv.euclides.
        runtime.ASTInformation("Vv", "", "Ptest", 7, 2, new String[]
        {""}, new String[] {""}, new int[] {16}));
5 }

```

Listing 4.6: This listing shows the class *Variables* holding the variable *Vv* defined in a var block of the example *Euclides* program.

```

1 package ecs;
2
3 public class Constants {
4     public static cgv.euclides.runtime.types.primitive.IntegerType
        Cc = Main.getFactory().createConstantInteger(new cgv.
        euclides.runtime.ASTInformation("Cc", "", "Ptest", 4, 2, new
        String[] {""}, new String[] {""}, new int[] {16, 16}), "1")
5 }

```

Listing 4.7: This listing shows the class *Constants* holding the constant *Cc* defined in a const block of the example program.

Having generated the Java files out of a *Euclides* program, the other files needed for execution are discussed. All generated java files associated with the example program can be found in Listing 4.8. One can see the already mentioned files *ECSCCompilerController.java* and *ECSCCompilerFactory.java* with it's respective interfaces *Controller.java* and *Factory.java*. *ASTInformation.java* as well as all interfaces for operators and types with their respective implementations *ECSCCompilerOperators.java* and *ECSCCompilerTypes.java* can also be found. All files mentioned above are part of the runtime environment needed for execution. The class *ECSCCompilerOperators* holds subclasses for all operators defined on the implemented runtime data types. For example the operator *and* is defined for the following type combinations:

- boolean, boolean
- boolean, error
- error, boolean
- error, error

```

euclides\runtime\ASTInformation.java
euclides\runtime\Controller.java
euclides\runtime\Factory.java
euclides\runtime\implementation\ECSCompilerController.java
euclides\runtime\implementation\ECSCompilerFactory.java
euclides\runtime\implementation\ECSCompilerOperators.java
euclides\runtime\implementation\ECSCompilerTypes.java
euclides\runtime\operators\OperatorAnd.java
euclides\runtime\operators\OperatorAssign.java
euclides\runtime\operators\OperatorEqual.java
euclides\runtime\operators\OperatorGreaterEqual.java
euclides\runtime\operators\OperatorGreaterThan.java
euclides\runtime\operators\OperatorIn.java
euclides\runtime\operators\OperatorLessEqual.java
euclides\runtime\operators\OperatorLessThan.java
euclides\runtime\operators\OperatorMinus.java
euclides\runtime\operators\OperatorNot.java
euclides\runtime\operators\OperatorNotEqual.java
euclides\runtime\operators\OperatorOr.java
euclides\runtime\operators\OperatorPlus.java
euclides\runtime\operators\OperatorSizeof.java
euclides\runtime\operators\OperatorSlash.java
euclides\runtime\operators\OperatorStar.java
euclides\runtime\types\primitive\BooleanType.java
euclides\runtime\types\primitive\ErrorType.java
euclides\runtime\types\primitive\IntegerType.java
euclides\runtime\types\primitive\MatrixType.java
euclides\runtime\types\primitive\RealType.java
euclides\runtime\types\primitive\ReferenceType.java
euclides\runtime\types\primitive\StringType.java
euclides\runtime\types\primitive\VectorType.java
euclides\runtime\types\primitive\VoidType.java
euclides\runtime\types\structured\ArrayType.java
euclides\runtime\types\structured\RecordType.java
euclides\runtime\types\structured\SetType.java
euclides\runtime\types\Type.java
Constants.java
Variables.java
Methods.java
Main.java

```

Listing 4.8: This listing shows the list of generated java files for the example program. One can see the runtime files need for execution as well as the transpiled Java files.

Runtime types are implemented as subclasses of the *ECSCompilerTypes* class. *Euclides* types are mapped to Java types using the following scheme:

- Boolean: Boolean
- Error: Long
- Integer: Long
- Matrix: ArrayList<ArrayList<Double>>
- Real: Double
- Reference: Object
- String: String
- Vector: ArrayList<Double>
- Void: Void
- Array: ArrayList<ECSType<?>>
- Set: HashSet<ECSType<?>>
- Record: ArrayList<ECSType<?>>

Basic functionality of all runtime types is defined in an abstract template class called *ECSType*. The methods *getValue*, *setValue*, *enterScope*, *exitScope* and *getSize* are common to all runtime types. Each concrete type class extends the abstract class *ECSType* and implements its interface.

In order to be able to compile the Java runtime files they are put together into one class called *Runtime*. This class holds all the Java classes as strings. A special creator class called *RuntimeCreator* generates this class. It is necessary to call the creator each time changes to the runtime environment are made in order to reflect the changes. In the *ECSCompiler* the Strings of the *Runtime* class are put together in an ArrayList together with the generated Java files from the Java transpiler. In a next step this ArrayList is processed by the Java compiler to generate class files. Finally these class files are packed into a jar file in order to be executed.

Chapter 5

Future Work

In this work, an IDE for procedural modeling including a new programming language called *Euclides* has been presented. Views for source code analysis, refactoring and interpretation form the tools to generate meaningful output. Currently executable output is generated in the form of Java code. However, there are a number of ideas to implement new output representations as well as to extend the programming language *Euclides* with new language constructs.

5.1 GML integration

The GML is a very simple stack based programming language. In combination with its OpenGL-based runtime engine the GML can be seen as a viewer with an integrated modeler. The idea is to integrate the GML as an additional target language for the IDE. This would enable the generation of GML code out of a *Euclides* program. The advantages of such an approach would be to following:

- The possibility to output 3D objects would be introduced to the IDE.
- A more beginner friendly programming language would be available for the GML.

5.2 Maya integration

Another possibility to generate output would be the integration in Maya. Maya is a 3D modeling software package used for architectural visualization and design. The idea is to develop a plug-in that enables Maya to use *Euclides* as an output format. One would be able to create 3D models using a sophisticated modeling environment while experiencing the benefits of procedural techniques.

5.3 New language constructs

The programming language *Euclides* incorporates a variety of language constructs. During the creation of this work some ideas were born to make the language more flexible. A first idea is to introduce a new data type to the language. It is a special data type for which type checking is partially disabled.

Another idea is to allow the definition of methods as data types. This would enable methods to be assigned to variables. It would make the language more flexible.

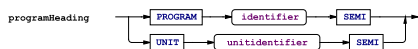
Chapter 6

Addendum

program



programHeading



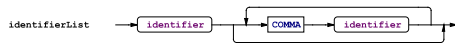
usesUnitsPart



identifier



identifierList



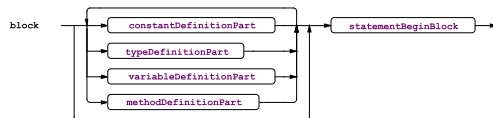
unitidentifier



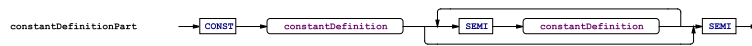
unitidentifierList



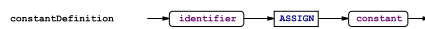
block



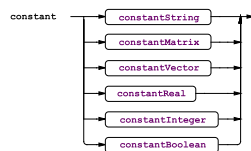
constantDefinitionPart



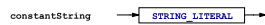
constantDefinition



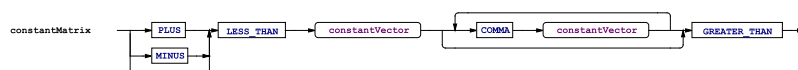
constant



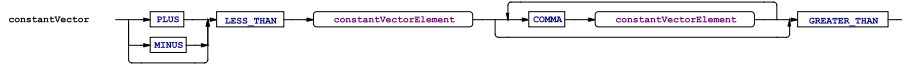
constantString



constantMatrix



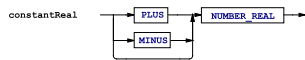
constantVector



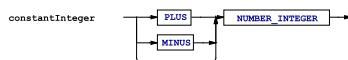
constantVectorElement



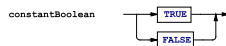
constantReal



constantInteger



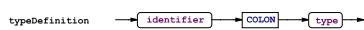
constantBoolean



typeDefinitionPart



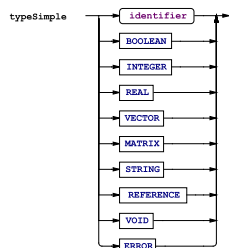
typeDefinition



type



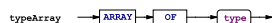
typeSimple



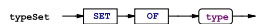
typeStructured



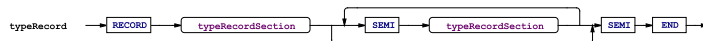
typeArray



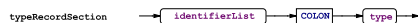
typeSet



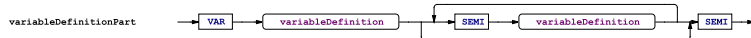
typeRecord



typeRecordSection



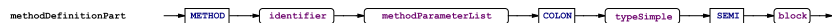
variableDefinitionPart



variableDefinition



methodDefinitionPart



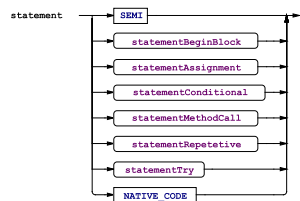
methodParameterList



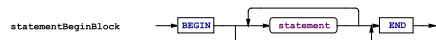
methodParameterSection



statement



statementBeginBlock



statementAssignment



statementMethodCall



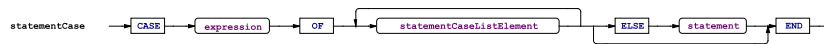
statementConditional



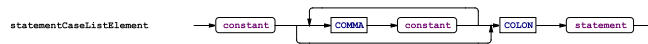
statementIf



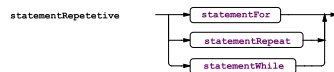
statementCase



statementCaseListElement



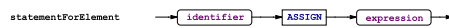
statementRepetitive



statementFor



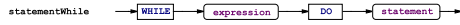
statementForElement



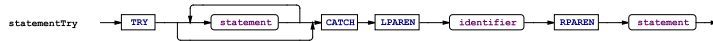
statementRepeat



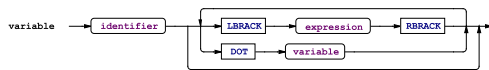
statementWhile



statementTry



variable



expression



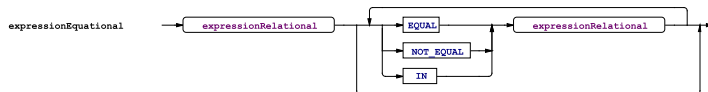
expressionLogical



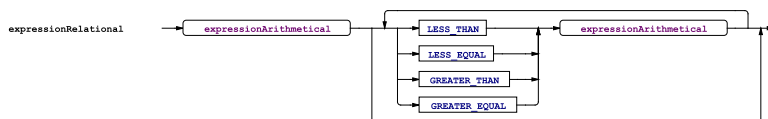
expressionLogicalAnd



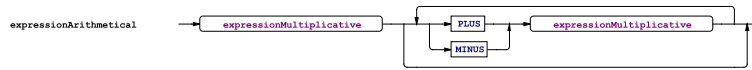
expressionEquational



expressionRelational



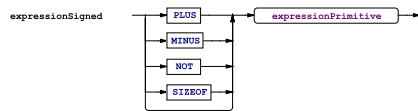
expressionArithmetical



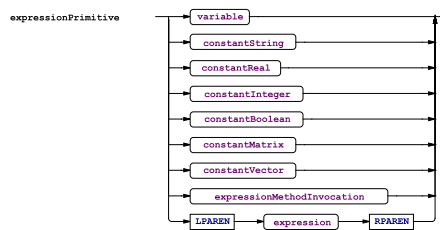
expressionMultiplicative



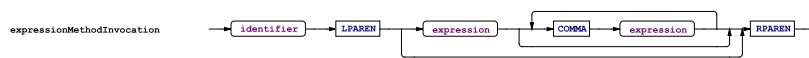
expressionSigned



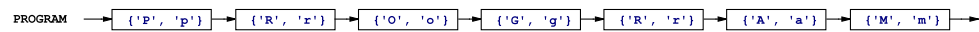
expressionPrimitive



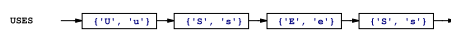
expressionMethodInvocation



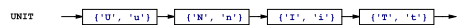
PROGRAM



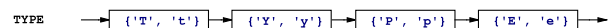
USES



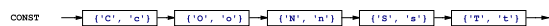
UNIT



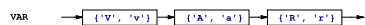
TYPE



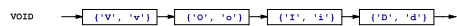
CONST



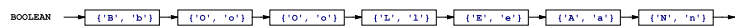
VAR



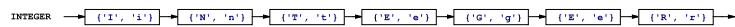
VOID



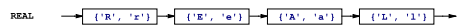
BOOLEAN



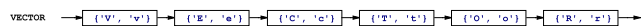
INTEGER



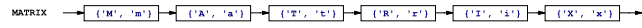
REAL



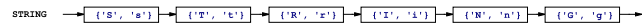
VECTOR



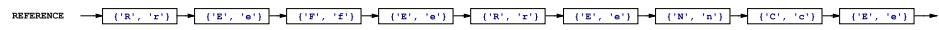
MATRIX



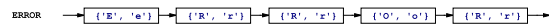
STRING



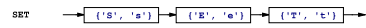
REFERENCE



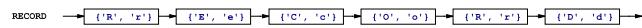
ERROR



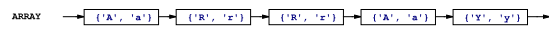
SET



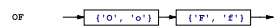
RECORD



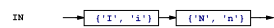
ARRAY



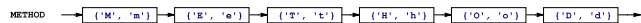
OF



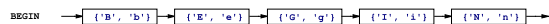
IN



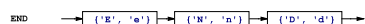
METHOD



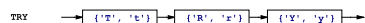
BEGIN



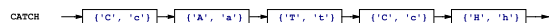
END



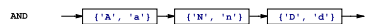
TRY



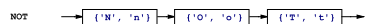
CATCH



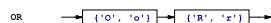
AND



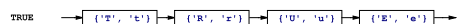
NOT



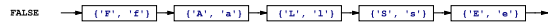
OR



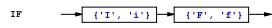
TRUE



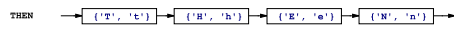
FALSE



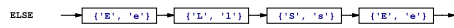
IF



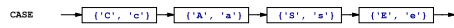
THEN



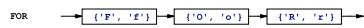
ELSE



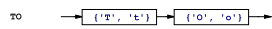
CASE



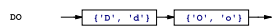
FOR



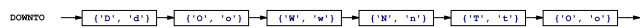
TO



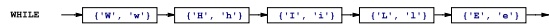
DO



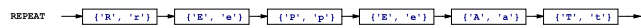
DOWNTO



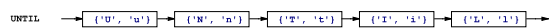
WHILE



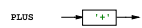
REPEAT



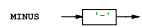
UNTIL



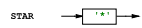
PLUS



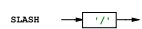
MINUS



STAR



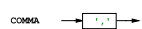
SLASH



ASSIGN



COMMA



SEMI

SEMI → [';'] →

COLON

COLON → [':'] →

EQUAL

EQUAL → ['='] →

NOT_EQUAL

NOT_EQUAL → ['!'] →

LESS_THAN

LESS_THAN → ['<'] →

LESS_EQUAL

LESS_EQUAL → ['<'] →

GREATER_EQUAL

GREATER_EQUAL → ['>'] →

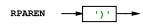
GREATER_THAN

GREATER_THAN → ['>'] →

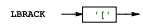
LPAREN

LPAREN → ['('] →

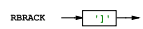
RPAREN



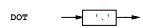
LBRACK



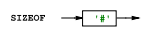
RBRACK



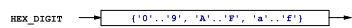
DOT



SIZEOF



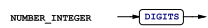
HEX_DIGIT



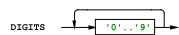
NUMBER_REAL



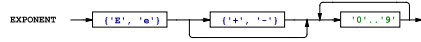
NUMBER_INTEGER



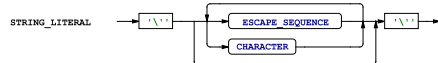
DIGITS



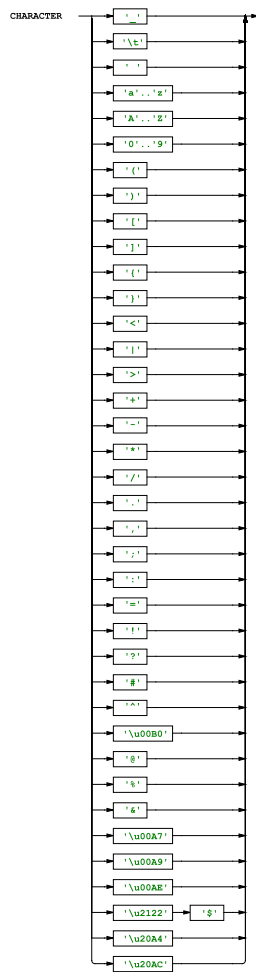
EXPONENT



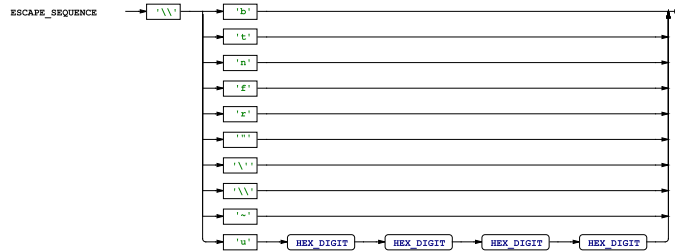
STRING_LITERAL



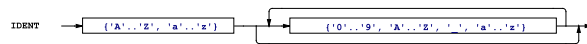
CHARACTER



ESCAPE_SEQUENCE



IDENT



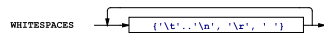
NATIVE_CODE



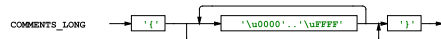
ANNOTATION



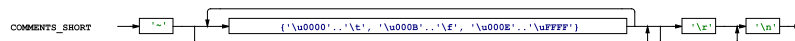
WHITESPACES



COMMENTS_LONG



COMMENTS_SHORT



List of Figures

1.1	GML example output	3
3.1	Data flow diagram of a parser	10
3.2	Example parse tree	13
3.3	Euclides source code refactoring	35
4.1	Euclides source code analysis	39

Bibliography

- [1] A. Day, D. Arnold, S. Havemann, and D. Fellner. Combining polygonal and subdivision surface approaches to modeling urban environments. *Cyberworlds, 2003. Proceedings. 2003 International Conference on*, pages 189–197, Dec. 2003.
- [2] O. Deussen and B. Lintermann. A modelling method and user interface for creating plants. *Proceedings of the conference on Graphics interface '97*, pages 189–197, 1997.
- [3] D. Finkensteller. Detailed building facades. *Computer Graphics and Applications, IEEE*, 28(3):58–66, May-June 2008.
- [4] D. Frijters and A. Lindenmayer. A model for the growth and flowering of aster novae-angliae on the basis of table $\langle 1, 0 \rangle$ l-systems. *L Systems*, pages 24–52, 1974.
- [5] B. Ganster and R. Klein. An integrated framework for procedural modeling. *Proceedings of Spring Conference on Computer Graphics 2007 (SCCG 2007)*, 23:150–157, 2007.
- [6] S. Havemann and D. Fellner. Generative parametric design of gothic window tracery. *Shape Modeling Applications, 2004. Proceedings*, pages 350–353, June 2004.
- [7] C. M. Hoffmann and K. J. Kim. Towards valid parametric cad models. *Computer-Aided Design*, 33(1):81 – 90, 2001.
- [8] A. Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, March 1968.
- [9] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, 2008.
- [10] E. Mendez, G. Schall, S. Havemann, D. Fellner, D. Schmalstieg, and S. Junghanns. Generating semantic 3d models of underground in-

- frastructure. *Computer Graphics and Applications, IEEE*, 28(3):48–57, May-June 2008.
- [11] P. Müller, P. Wonka, S. Haegler, U. Andreas, and L. Van Gool. Procedural modeling of buildings. *Proceedings of 2006 ACM Siggraph*, 25(3):614–623, 2006.
 - [12] P. Müller, G. Zeng, P. Wonka, and L. Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, 28(3):1–9, 2007.
 - [13] A. Paoluzzi. Generative geometric modeling in a functional environment. *Design and Implementation of Symbolic Computation Systems*, (1128):79–97, 1996. Invited lecture.
 - [14] Y. Parish and P. Müller. Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 28:301–308, 2001.
 - [15] T. Parr. ANTLR, ANTLR Parser Generator.
online: <http://www.antlr.org>.
 - [16] T. Parr. StringTemplate Template Engine.
online: <http://www.stringtemplate.org>.
 - [17] P. Prusinkiewicz and J. Hanan. Visualization of botanical structures and processes using parametric l-systems. *Scientific visualization and graphics simulation*, pages 183–201, 1990.
 - [18] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.